

OCR Computer Science A Level

2.1.2 Thinking Ahead

Advanced Notes



Specification:

2.1.2 a)

- Identify the inputs and outputs for a given situation.

2.1.2 b)

- Determine the preconditions for devising a solution to a problem.

2.1.2 c)

- The nature, benefits and drawbacks of caching

2.1.2 d)

- The need for reusable program components



Inputs and Outputs

Designing a solution entails thinking ahead about the **different components of a problem** and how they will be **handled in the best way** possible. Thinking ahead allows developers to consider problems or difficulties that may arise when the software is used. Taking these factors into account at an early stage, developers can design strategies to make programs **easy and intuitive to use**.

At their core, all computational problems consist of inputs which are processed to produce an output. **Inputs** include any **data that is required to solve the problem**, entered into the system by the user. Often, the **order** in which data is input and the **method** of input must also be taken into consideration. **Outputs** are the **results that are passed back** once the inputs have been processed and the problem solved. Designers must decide on a **suitable data type, structure and method** to use in order to present the solution, given the scenario.

Note

Information does not necessarily have to be output on a screen or monitor. Output data includes information in a visual or audio format, which may require an output device such as a printer or speaker.

You must be able to identify the inputs and outputs that would be required to form a program given a scenario. Take, for example, a program designed for an ATM.

| Inputs | Outputs |
|---|--|
| Transaction type: Deposit? Balance check? Withdrawal? | If deposit selected: Display total amount entered on screen |
| Card details, captured using a card reader | If balance check selected: Display total account balance on screen |
| PIN, entered via keypad | If withdrawal selected: Dispense correct amount of cash. |
| | Print receipt to confirm transaction |
| | Speaker provides verbal feedback throughput. |

You might also be asked about **how this data is captured**, or is **relayed back** to the user once processed. The input devices required would be a touch screen, magnetic stripe card reader and keypad, while output devices would include a monitor, cash dispenser, printer and speakers. Typically, designers begin by considering what outputs are required of the solution based on the user's requirements. The next step is to identify the inputs required and how these need to be processed to achieve these outputs.



Preconditions

Preconditions are requirements which must be met before a program can be executed. If the preconditions are not met, the program will fail to execute or return an invalid answer. Specifying preconditions means that a subroutine can safely expect the arguments passed to it to meet certain criteria, as defined by the preconditions. Preconditions can be tested for within the code but are more often included in the documentation accompanying a particular subroutine, library or program.

Consider, for example, the function `pop()`, which removes the last item added to a stack:

```
function pop():
    if top = 0 then
        print "No items in the stack."
    else:
        element = stack[top]
        top = top-1
    endif
endfunction
```

The function first checks that the stack is not empty by checking whether the top pointer, `top`, is greater than 0. Clearly without this condition being met, no items can be removed from the stack. It is important that this is tested for within the code, as popping from an empty stack would otherwise produce an error that would cause the program to crash.

Preconditions can also be included within the documentation, in which case it is the user's responsibility to ensure inputs meet the requirements specified by these preconditions. A common example of this is the factorial function, which can only be called upon positive numbers. Rather than checking that the arguments passed to the function must be non-negative, this is specified within the documentation accompanying this function. Including preconditions within the documentation reduces the length and complexity of the program as well as saving time needed to debug and maintain a longer program.

Arguments

The values passed into the parameters in a subroutine.

The purpose of preconditions is to ensure that the necessary checks are carried out before the execution of a subroutine, either by the user or as part of the subroutine. By explicitly ensuring these conditions are met, subroutines are made more reusable.



Caching

Caching is the process of **storing instructions or values in cache memory** after they have been used, as they **may be used again**. This **saves time** which would have been needed to store and retrieve the instructions from secondary storage again. Caching is very common in the **storage of web pages**. The web pages that a user frequently accesses are cached, so the next time one of these pages is accessed, content can be **loaded without any delay**. This also means images and text do not have to be downloaded again multiple times, **freeing up bandwidth** for other tasks on a network.

A more advanced variation of caching and thinking ahead is **prefetching**, in which **algorithms predict which instructions are likely to soon be fetched**. The instructions and data which are likely to be used are then **loaded and stored in cache before they are fetched**. By thinking ahead, therefore, **less time is spent waiting** for instructions to be loaded into RAM from the hard disk.

Clearly, one of the biggest limitations to this is the **accuracy of the algorithms used** in prefetching, as they can **only provide an informed prediction** as to the instructions which are likely to be used and there is no guarantee that this will be right. Similarly, the effectiveness of caching depends on how well a caching algorithm is able to manage the cache. Larger caches still take a long time to search and so **cache size limits how much data can be stored**. In general, this form of thinking ahead can be **difficult to implement** but can significantly improve performance if implemented effectively.

Reusable Program Components

Commonly used functions are often **packaged into libraries for reuse**. Teams working on large projects that are likely to make use of certain components in multiple places might also choose to put together a library so these components can be reused. Reusable components include **implementations of abstract data structures** such as queues and stacks as well as **classes** and **subroutines**. When designing a piece of software, the problem is **decomposed**: it is **broken down into smaller, simpler tasks**. This allows developers to think ahead about how each task can be solved, and identify where **program components developed in the past**, or **externally-sourced program components**, can be **reused** to simplify the development process.

Decomposition

Decomposition is a **computational method** used in **problem-solving**. It is discussed in detail in **2.2.2**.



Reusable components are **more reliable** than newly-coded components, as they have **already been tested** and any bugs dealt with. This **saves time, money and resources**. Subroutines can then simply be reused with different arguments to produce a variety of outputs. Producing well-tested, reusable components means that they can also be **reused in future projects, saving development costs**. However, it may not always be possible to integrate existing components developed by third parties due to **compatibility issues** with the rest of the software. This may mean these components need to be modified to work with existing software, which can sometimes be more costly and time-consuming than developing them in-house.

